Python – Class III (Part - 2)

bytes, bytearray, range, list, tuple, set, frozenset, dict, None

LavaTech Technology

Bits, Nibbles, and Byte

Each 1 or 0 in a binary number is called a bit.

From there, a group of 4 bits is called a nibble

8-bits makes a byte.

Length	Name	Example
1	Bit	0
4	Nibble	1011
8	Byte	10110101

What is Ascii Code

- ASCII (American Standard Code for Information Interchange) is the most common format for text files in computers and on the Internet.

- In an ASCII file, each alphabetic, numeric, or special character is represented with a 7-bit binary number (a string of seven 0s or 1s).

- 128 possible characters are defined.

Ascii chart

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

What is Unicode?

- Lets say Joe is a software developer.
- He says that he will only ever need English, and as such only wants to use ASCII.
- This might be fine for Joe the user, but this is not fine for Joe the software developer. Approximately half the world uses non-Latin characters and using ASCII is not sufficient as it does not contain other letters.
- Thus came Unicode. It assigns every character a unique number called a code point.
- **Memory considerations**
  - So how many bytes give access to what characters in these encodings?
    - UTF-8:
      - 1 byte: Standard ASCII
      - 2 bytes: Arabic, Hebrew, most European scripts (most notably excluding Georgian)
      - 3 bytes: BMP
      - 4 bytes: All Unicode characters
    - UTF-16:
      - 2 bytes: BMP
      - 4 bytes: All Unicode characters

- If you'll be working mostly with ASCII characters, then UTF-8 is certainly more memory efficient.
- However, if you're working mostly with non-European scripts, using UTF-8 could be up to 1.5 times less memory efficient than UTF-16.
- When dealing with large amounts of text, such as large web-pages or lengthy word documents, this could impact performance.

Relation of Python & 1 Byte

Python by default keeps all the ascii character of a byte size (8bits) initialized in memory.

Now what does that mean? :(

```
>>> bin(0)
'0b0'
>>>
>>> bin(127)
'0b1111111'
```

Ascii characters are of 7 bits -

Notice -
Binary value for 0 is 0
Binary value for 127 is 1111111

Notice -
Binary value for 255 is 11111111
Binary value for 127 is 1111111

```
>>> bin(256)
'0b100000000'
>>>
>>> bin(255)
'0b11111111'
```

- By default, Python will reserve all ascii codes in memory and assign them fix address, i.e 0-127 characters will have fix memory address which wont change throughout the life of program!
- What about remaining 128-256 characters?
- Note that ascii code are not defined for 128-256, so that does’nt mean that python will not entertain 128-256 (remember that this range is of 8bits with is 1 byte)
- So python will initialise all ascii code and 128-256 numbers in memory from very beginning regardless of whether it is used or not!

Let’s see this practically

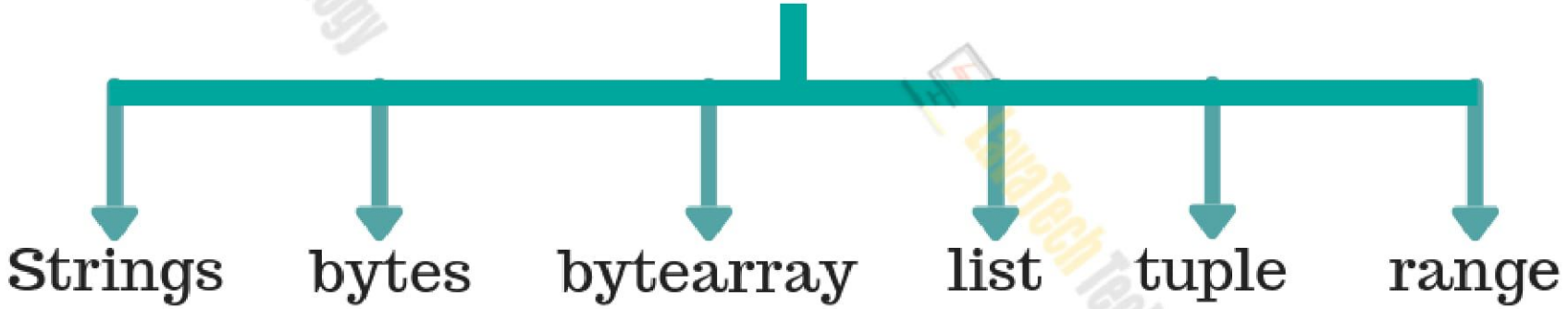
```
>>> c=127
>>> id(127)
139655390554368
>>>
>>> id(c)
139655390554368
>>>
>>> d=127
>>> id(d)
139655390554368
>>>
>>> id(255)
139655390558464
>>>
>>> a=255
>>> id(a)
139655390558464
>>>
>>> e=256
>>> id(e)
139655390558496
>>>
>>> id(256)
139655390558496
>>>
>>> f=256
>>> id(f)
139655390558496
>>>
```

```
>>> a=1000
>>> id(a)
140289546425264
>>>
>>> b=1000
>>> id(b)
140289546425168
>>>
>>> c=300
>>> id(c)
140289546425328
>>>
>>> f=300
>>> id(f)
140289546425360
```

```
>>> a="!"
>>> id(a)
140289672400320
>>>
>>> b="!"
>>> id(b)
140289672400320
>>>
>>> a="k"
>>> id(a)
140289672748704
>>>
>>> b="k"
>>> id(b)
140289672748704
>>>
>>> c="~"
>>> id(c)
140289672942736
>>>
>>> f="~"
>>> id(f)
140289672942736
```

LavaTech Technology

Sequences DataType in Python



Bytes DataType

- Bytes objects contain single bytes ie. 8bits values
- There are 3 syntax to create byte datatype -

(1) To store a string as byte datatype -

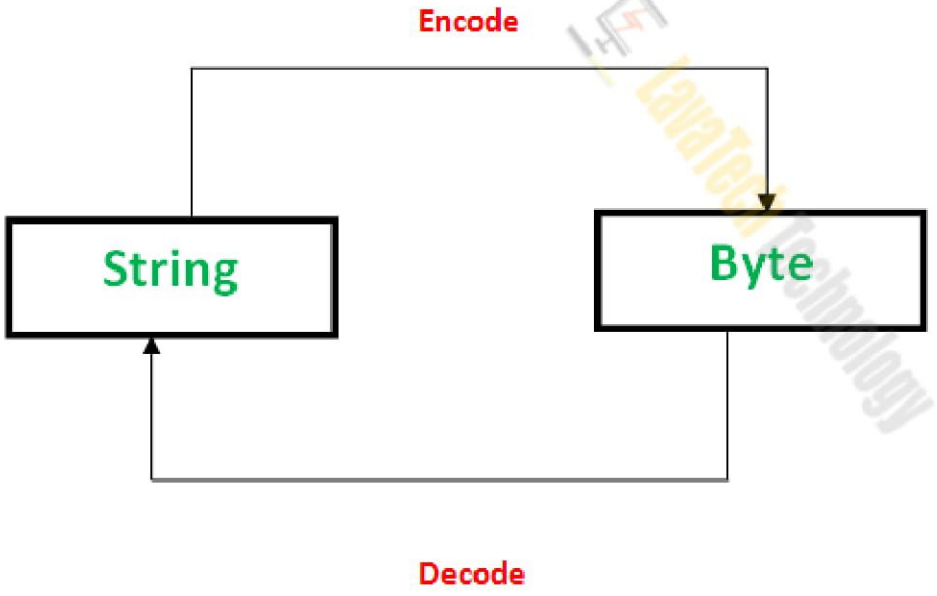
If the string contains only ascii character -

- var=b'string' or var=str.encode('ascii') or var=bytes(str,'ascii')

If the string containes unicode characters -

- str='\uCode'
- var=str.encode() or var=str.encode('utf-8') or var=bytes(str,'utf-8')
- To display output → var.decode()

- Byte objects are sequence of Bytes, whereas Strings are sequence of characters.
- Byte objects are in machine readable form internally, Strings are only in human readable form.
- Since Byte objects are machine readable, they can be directly stored on the disk. Whereas, Strings need encoding before which they can be stored on disk.



Practicle

```
>>> var=b'python'
>>> type(var)
<class 'bytes'>
>>>
>>> var
b'python'
>>> var.decode()
'python'
>>>
>>> var[0]
112
>>> var[1]
121
>>> chr(112)
'p'
>>> chr(121)
'y'
>>>
>>>
>>> var[0:3]
b'pyt'
>>> var[0:3].decode()
'pyt'
>>>
```

```
>>> var[0].decode()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'decode'
>>>
>>> type(var[0])
<class 'int'>
```

Chr() function is used to convert ascii numbers into ascii characters

Practicle

```
>>>
>>> new='\u039B'
>>> new
'Λ'
>>>
>>> var2=new.encode()
>>> var2
b'\xce\x9b'
>>>
>>> var.decode()
'Λ'
>>>
>>> var[0]
206
>>> var[1]
155
>>> var[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index out of range
>>>
>>> chr(206)
'Î'
>>> chr(155)
'\x9b'
```

Practicle

```
>>> var='apple'
>>> b=bytes(var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string argument without an encoding
>>>
>>> b=bytes(var,'ascii')
>>>
>>> b
b'apple'
>>>
>>> var='\u039B'
>>>
>>> b=bytes(var,'utf-8')
>>> b
b'\xce\x9b'
>>>
>>> b.decode()
'Λ'
```

Syntax - 2

- If you wish to store numbers as byte datatype, then supply numbers as a “list”.
- As bytes datatype can store only 1byte size data, it means it can only store number from 0-255

  Synxtax -

  list1=[1,2,3]

  var=bytes(list1)

Practicle

```
>>> l1=[40,50,100]
>>> var=bytes(l1)
>>>
>>> type(l1)
<class 'list'>
>>> type(var)
<class 'bytes'>
>>>
>>> var
b'(2d'
>>>
>>> var.decode()
'(2d'
>>>
```

```
>>> ord("(")
40
>>> ord("2")
50
>>> ord("d")
100
>>>
>>> var[0]
40
>>> var[1]
50
>>> var[2]
100
>>> var[-1]
100
>>> var[0:2]
b'(2'
>>>
```

Ord() function is used to convert ascii character into ascii integers

Note – The error faced in case the value is more than 256 →

```
>>> l1=[40,50,1000]
>>> var=bytes(l1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: bytes must be in range(0, 256)
>>>
```

Syntax - 3

- Create a byte of given integer size
- Syntax -

  var=bytes(integer)

  Example -

  var=bytes(3)

```
>>> var=bytes(4)
>>> type(var)
<class 'bytes'>
>>>
>>> var
b'\x00\x00\x00\x00'
>>> var[0]
0
>>> var[1]
0
>>> var[2]
0
>>> var[3]
0
```

Bytes are immtable

```
>>> a='apple'
>>> b=bytes(a,'ascii')
>>> b.decode()
'apple'
>>>
>>> b[0]
97
>>> b[0]=90
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
>>>
>>> a=[100,200,150]
>>> b=bytes(a)
>>> b
b'd\xc8\x96'
>>> b[0]
100
>>> b[0]=130
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
>>>
```

What is Practicle usage of bytes datatype?

- In Python 3, strings are Unicode, but when transmitting on the network, the data needs to be bytes strings instead.

```
[root@forwarder ~]# cat server.py
import socket
s = socket.socket()
print ("Socket successfully created")
s.bind(('127.0.0.1', 12345))
print ("socket binded to 12345")
s.listen(5)
print("socket is listening")
while True:
   c, addr = s.accept()
   print('Got connection from', addr)
   c.send(b'Thank you for connecting')
   c.close()
[root@forwarder ~]#
```

```
[root@forwarder ~]# python3 server.py
Socket successfully created
socket binded to 12345
socket is listening
Got connection from ('127.0.0.1', 43089)
Got connection from ('127.0.0.1', 43090)
Got connection from ('127.0.0.1', 43093)
```

```
[root@forwarder ~]# nc 127.0.0.1 12345
Thank you for connecting

Ncat: Broken pipe.
[root@forwarder ~]# nc 127.0.0.1 12345
Thank you for connecting

Ncat: Broken pipe.
[root@forwarder ~]# nc 127.0.0.1 12345
Thank you for connecting
```

Bytearray Datatype

- Bytearray is same as bytes datatype.
- The bytearray() method returns a bytearray object which is a mutable (can be modified) sequence of integers in the range 0 <=x < 256.
- If you want the immutable version, use bytes() method.

Practicle

```
>>> a=bytearray(3)
>>> a
bytearray(b'\x00\x00\x00')
>>> a.decode()
'\x00\x00\x00'
>>>
>>> str='apple'
>>> a=bytearray(str,'ascii')
>>> a
bytearray(b'apple')
>>> a.decode()
'apple'
>>>
>>> l1=[100,20,40]
>>> a=bytearray(l1)
>>> a
bytearray(b'd\x14(')
>>> a.decode()
'd\x14('
>>>
>>> str='\u039B'
>>> a=bytearray(str,'utf-8')
>>> a
bytearray(b'\xce\x9b')
>>> a.deocde()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytearray' object has no attribute 'deocde'
>>> a.decode()
'Λ'
>>>
```

Bytearray is mutable

```
>>> a='apple'
>>> b=bytearray(a,'ascii')
>>> b
bytearray(b'apple')
>>> b.decode()
'apple'
>>> b[0]
97
>>> b[0]=90
>>> b.decode()
'Zpple'
>>>
>>> a=[5,10,15,20]
>>> b=bytearray(a)
>>> b
bytearray(b'\x05\n\x0f\x14')
>>> b[0]
5
>>> b[0]=0
>>> b[0]
0
>>> b[0:5]
bytearray(b'\x00\n\x0f\x14')
>>> b[0:5].decode()
'\x00\n\x0f\x14'
>>>
```

```
>>>
>>> a=bytearray(5)
>>> a[0]
0
>>> a[0]=5
>>> a[0]
5
>>> a[1]=10
>>> a[0]
5
>>> a[1]
10
>>>
>>> for i in a:
...     print(i)
...
5
10
0
0
0
```

Range function

- The range() type returns an immutable sequence of numbers between the given start integer to the stop integer.
- range() constructor has two forms of definition:
  - range(stop)
  - range(start, stop[, step])
    - start - integer starting from which the sequence of integers is to be returned
    - stop - integer before which the sequence of integers is to be returned. The range of integers end at stop – 1.
    - step (Optional) - integer value which determines the increment between each integer in the sequence
- range(stop)
  - Returns a sequence of numbers starting from 0 to stop - 1
  - Returns an empty sequence if stop is negative or 0.

How range works?

- # empty range
- print(list(range(0)))

- # using range(stop)
- print(list(range(10)))

- # using range(start, stop)
- print(list(range(1, 10)))

```
>>>
>>> a=range(0,10)
>>>
>>> a
range(0, 10)
>>>
>>> list(a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> a[0]
0
>>>
>>> a[0]=10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support item assignment
>>>
```

```
>>>
>>> a=range(2,-14,-2)
>>>
>>> list(a)
[2, 0, -2, -4, -6, -8, -10, -12]
>>>
```

List Data Type

- List is an **ordered sequence** of items. It is one of the most used datatype in Python and is very flexible. All the items in a list **do not need to be of the same type**. **Duplicates values** are allowed and list is **growable in nature**.
- Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

  Example – a=[“apple”,1.0,200]
- We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts form 0 in Python.
- We can use append() & remove() to add/remove more members to list.

  Example – a.append(‘test’) , a.remove(‘test’)

```
>>> a=[5,10,15,20,25,30,35,40]
>>>
>>> a[2]
15
>>>
>>> a[2]="apple"
>>>
>>> a
[5, 10, 'apple', 20, 25, 30, 35, 40]
>>>
>>> a[0:3]
[5, 10, 'apple']
>>>
>>> a[5:]
[30, 35, 40]
>>>
>>> a[-1]
40
>>>
```

```
>>>
>>> a=["lava","tech"]
>>>
>>> a.append("technology")
>>>
>>> a
['lava', 'tech', 'technology']
>>>
>>> a.remove('tech')
>>>
>>> a
['lava', 'technology']
>>>
```

Tuple DataType

- Tuple is an ordered sequence of items same as list.The only difference is that tuples are immutable. Tuples once created cannot be modified.
- Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.
- It is defined within parentheses () where items are separated by commas.

  Example → t = (5,'program', 1+3j)

  We can use the slicing operator [] to extract items but we cannot change its value.

```
>>> a=(5,"python",1+3j)
>>>
>>> a
(5, 'python', (1+3j))
>>>
>>> a[-1]
(1+3j)
>>>
>>> a[1:]
('python', (1+3j))
>>>
>>> a[1]=50
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

Set DataType

- Set is an unordered collection of unique items.
- Set is defined by values separated by comma inside braces { }.

  Example → a= {5,10,15,20,”apple”}
- Items in a set are not ordered.
- We can perform set operations like union, intersection on two sets.
- Set have unique values. They eliminate duplicates.
- Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [] does not work.

```
>>>
>>> data={5,10,15,"apple"}
>>>
>>> data
{10, 5, 'apple', 15}
>>>
>>> type(data)
<class 'set'>
>>>
>>> data[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>>
>>> data={10,10,10,10,5,5,5,2,2,"apple","apple"}
>>>
>>> data
{10, 2, 5, 'apple'}
>>>
```

- Set can grow or compress using add() & remove() functions.
- Example -

  s={“don”,”ricki”,’shanti’,’banti’}

  s.add(“kavi”)

  s.remove(“ricki”)

```
>>> s={"don","ricki","shanti","banti"}
>>> s
{'ricki', 'banti', 'don', 'shanti'}
>>>
>>> s.add("kavi")
>>>
>>> s
{'kavi', 'banti', 'ricki', 'don', 'shanti'}
>>>
>>> s.remove("don")
>>>
>>> s
{'kavi', 'banti', 'ricki', 'shanti'}
>>>
```

Frozenset Datatype

- Frozenset is just like set datatype, except, frozenset are immutable.
- Example -
- s={“don”,”ricki”,’shanti’,’banti’}
- new=frozenset(s)
- new.add(“apple”) ← Error

```
>>> s={"don","ricki","shanti","banti"}
>>>
>>> new=frozenset(s)
>>>
>>> new
frozenset({'ricki', 'banti', 'don', 'shanti'})
>>>
>>> new.add("apple")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>>
>>> new.remove("don")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'remove'
>>>
```

Dict DataType

- Dictionary is an unordered collection of key-value pairs.
- It is generally used when we have a huge amount of data. We must know the key to retrieve the value.
- In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value.
- Key and value can be of any type.

```
>>>
>>> data={1:"apple",2:"mango",3:"grapges",'a':"apple",'m':"mango"}
>>>
>>> data[1]
'apple'
>>> data['a']
'apple'
>>>
>>> data[1]="jackfruit"
>>>
>>> data[1]
'jackfruit'
>>>
>>> data
{1: 'jackfruit', 2: 'mango', 3: 'grapges', 'a': 'apple', 'm': 'mango'}
>>>
>>> type(data)
<class 'dict'>
>>>
>>> data[3]="new"
>>>
>>> data
{1: 'jackfruit', 2: 'mango', 3: 'new', 'a': 'apple', 'm': 'mango'}
>>>
```

Python’s null equivalent: None

- The syntax to assign the None type to a variable, is very simple. As follows:

  my_none_variable = None

- Why Use Python's None Type?
- Often you will want to perform an action that may or may not work. Using None is one way you can check the state of the action later. Here's an example:

```
[root@forwarder ~]#
[root@forwarder ~]# cat test.py
database_connection = None
if database_connection is None:
    print('The database could not connect')
else:
    print('The database could connect')
[root@forwarder ~]#
[root@forwarder ~]# python3 test.py
The database could not connect
[root@forwarder ~]#
```

Another example of None

```
[root@forwarder ~]# cat test2.py
def f1(): a=10

def f2(): print("Hello there!")

f1()

f2()

print(f1())
print(f2())
[root@forwarder ~]#
[root@forwarder ~]# python3 test2.py
Hello there!
None
Hello there!
None
[root@forwarder ~]#
```